SIAM J. SCI. COMPUT.
Vol. 17, No. 2, pp. 454-478, March 1996
© 1996 Society for Industrial and Applied Mathematics
010

ALTERNATING-DIRECTION LINE-RELAXATION METHODS ON MULTICOMPUTERS*

JÖRN HOFHAUS† AND ERIC F. VAN DE VELDE‡

Abstract. We study the multicomputer performance of a three-dimensional Navier-Stokes solver based on alternating-direction line-relaxation methods. We compare several multicomputer implementations, each of which combines a particular line-relaxation method and a particular distributed block-tridiagonal solver. In our experiments, the problem size was determined by resolution requirements of the application. As a result, the granularity of the computations of our study is finer than is customary in the performance analysis of concurrent block-tridiagonal solvers. Our best results were obtained with a modified half-Gauss-Seidel line-relaxation method implemented by means of a new iterative block-tridiagonal solver that is developed here. Most computations were performed on the Intel Touchstone Delta, but we also used the Intel Paragon XP/S, the Parsytec SC-256, and the Fujitsu S-600 for comparison.

Key words. Navier-Stokes equations, concurrency, parallelism, block-tridiagonal systems, tridiagonal systems, ADI, alternating directions

AMS subject classifications. 65M20, 65N40, 65Y05, 76C05, 76M20

1. Introduction.

When using alternating-direction line-relaxation methods for systems of partial-differential equations discretized on a rectangular grid, one must solve many block-tridiagonal systems of linear equations in every relaxation step. This type of computation surfaces in many applications. In our work, we faced it when parallelizing a highly vectorized solver for the three-dimensional, unsteady, and incompressible Navier-Stokes equations [4] for use on multicomputers. In this paper, we shall discuss and analyze five line-relaxation methods and six distributed block-tridiagonal solvers used during the course of this project. We have measured the performance of several combinations of relaxation methods and block-tridiagonal solvers on three multicomputers (the Intel Touchstone Delta [5], the Intel Paragon XP/S [6], and the Parsytec SC-256 [13]) and on one conventional vector processor (the Fujitsu S-600).

Three-dimensional computations are more complex than two-dimensional ones because of additional coordinates for the geometry and the vector fields. Three-dimensional computations are also algorithmically different, because every line-relaxation step requires the solution of a larger number of smaller-sized linear systems than a comparable two-dimensional line-relaxation step. To see this, consider a two- and a three-dimensional problem of the same size, i.e., with the same number of unknowns. The two-dimensional problem on an $M \times M$ grid requires the solution of $O(M)$ block-tridiagonal systems of size $O(M)$, while the three-dimensional problem on an $N \times N \times N$ grid with $N^3 = M^2$ requires the solution of $O(N^2) = O(M^{4/3})$ systems of size $O(N) = O(M^{2/3})$.

On multicomputers, the reduced system size has a serious impact. There exist many concurrent algorithms to solve block-tridiagonal systems; see, e.g., [2], [10], [12], [14]-[16]. However, these methods have been studied mainly for very coarse-grain computations, i.e., for computations in which the ratio of the system size over the number of nodes is high. Bondeli [2] studied computations with less than 25 nodes and system sizes exceeding 16, 800. Krechel,

*Received by the editors August 16, 1993; accepted for publication (in revised form) October 24, 1994. This material is based upon work supported by the NSF under Cooperative Agreement No. CCR-9120008. Access to the Intel Touchstone Delta was provided by the Concurrent SuperComputing Consortium.

†Aerodynamisches Institut der RWTH Aachen, Wüllnerstr. zw. 5 u. 7, 52062 Aachen, Germany (jorn@aia.rwth-aachen.de). Caltech visit sponsored by the German Research Association (DFG) within the project: Strömungssimulation mit Hochleistungsrechnern.

‡Applied Mathematics 217-50, California Institute of Technology, Pasadena, CA 91125 (evdv@ama.caltech.edu).

Plum, and Stüben [14] improved the efficiency of a modified cyclic-reduction algorithm on the 16-node iPSC/2-VX. Their computations of moderate-to-coarse granularity solved 256 tridiagonal systems of size 128 and the highest-achieved efficiency was about 50%. For the reasons mentioned above, the granularity of the block-tridiagonal solver in our three-dimensional application is much smaller when running the code for a grid of fixed size on all 512 computing nodes of the Delta. Constructing an efficient concurrent program for this problem is a challenge.

An outline of the paper follows. In §2, a brief description of the implemented relaxation methods is given. In §3, we shall study algorithm and concurrency issues of several basic concurrent alternating-direction relaxation methods. Each method is based on a different distributed solver for block-tridiagonal systems. First, the sequential block-tridiagonal solver is distributed and pipelined to obtain a concurrent line-relaxation method. Subsequently, we use concurrent tridiagonal solvers based on recursive doubling, cyclic reduction, partitioning, and divide and conquer, respectively. Finally, a new iterative tridiagonal solver is developed. In §4, we analyze the performance on the Delta of the proposed methods applied to a Navier–Stokes solver. The Navier–Stokes equations and their discretization for an interesting application are given in §4.1. We give this description for the sole purpose of defining precisely how the performance data were obtained. The fluid-dynamical results of our computations are discussed in another paper [1]. In §5, computations on the Paragon, the Parsytec, and the Fujitsu are compared with those on the Delta.

2. Relaxation methods. The discretization of partial-differential equations often leads to a large sparse system of equations

$$A\vec{u} = \vec{f}. \tag{1}$$

Classical relaxation methods to solve (1) are obtained by splitting the coefficient matrix A into

$$A = G + H$$

with G an easily inverted matrix. This defines the iteration

$$G\vec{u}^{(\nu)} = \vec{f} - H\vec{u}^{(\nu-1)}, \tag{2}$$

which converges to the exact solution $\vec{u}^*$ if the spectral radius of $G^{-1}H$ is less than one; e.g., see [11].

In the description that follows, it is convenient to think of the three-dimensional Poisson problem discretized to second-order accuracy on a three-dimensional rectangular grid of size $M \times N \times K$. Then, interior grid points are identified by means of a triple (m, n, k) with $1 \le m \le M$, $1 \le n \le N$, and $1 \le k \le K$. One unknown and one equation is associated with each interior grid point.

One *elementary line-relaxation step* updates all unknowns of one grid line simultaneously. For example, an x-line is a set of grid points (m, n, k) where $1 \le m \le M$ and n and k are fixed. An x-line is, therefore, identified by means of a couple (n, k). An elementary x-line-relaxation step reduces to the tridiagonal system

$$a_m u_{m-1} + b_m u_m + c_m u_{m+1} = d_m, \qquad m = 1, \ldots, M \tag{3}$$

for each x-line. One *x-line-relaxation step* updates all x-lines of the grid once, i.e., it performs an elementary x-line-relaxation step for all (n, k) with $1 \le n \le N$ and $1 \le k \le K$. For three-dimensional problems, an *alternating-direction line-relaxation step* consists of an *x-line-relaxation step*, a *y-line-relaxation step*, and a *z-line-relaxation step*. We restrict our discussion to the x-line-relaxation step. In matrix-vector notation, the system (3) has the form

$$T\vec{u} = \vec{d}, \tag{4}$$

with

$$T = \begin{pmatrix} b_1 & c_1 & & & & & \\ a_2 & b_2 & c_2 & & & \mathbf{0} & \\ & \ddots & \ddots & \ddots & & & \\ & & \ddots & \ddots & \ddots & & \\ & \mathbf{0} & & a_{M-1} & b_{M-1} & c_{M-1} & \\ & & & & a_M & b_M & \end{pmatrix}.$$

The right-hand side vector $\vec{d}$ depends on the boundary conditions, which add the terms $a_1 u_0$ and $c_M u_{M+1}$ to the right-hand side of the first and the last equation, respectively. The right-hand side vector also depends on neighboring x-lines $(n+1, k)$, $(n-1, k)$, $(n, k-1)$, and $(n, k+1)$. (This assumes a classical seven-point stencil for second-order discretization of the Poisson equation in three dimensions.) Three variants of the line-relaxation method were implemented.

When the right-hand side terms for x-line (n, k) are computed exclusively with u-values that were known before the x-line-relaxation step began, the method is called *Jacobi line relaxation.* In this case, all elementary x-line-relaxation steps and all systems (3) are independent. In principle, all tridiagonal solves may be performed concurrently provided that the coefficients of T are not distributed over several processes.

Gauss–Seidel line relaxation assigns an order to the x-lines and uses the updated u-values obtained in preceding elementary x-line-relaxation steps to compute the right-hand sides of the current elementary x-line-relaxation step. *Lexicographic ordering* approach enforces a rigid and sequential ordering on the solution of the tridiagonal systems: (n, k) follows $(n-1, k)$ and $(n, k-1)$.

For the purpose of vectorization and/or concurrent computing, it makes sense to make many tridiagonal systems independent of one another so that they can be solved simultaneously. *Red-black ordering* performs elementary line-relaxation steps first for all x-lines for which $n+k$ is even and, subsequently, for all x-lines for which $n+k$ is odd.

Lexicographic ordering may be kept provided the relaxation method is modified. For the x-line-relaxation step, e.g., the systems of grid plane $k =$ constant are made independent of those of grid plane $k-1$ by using the old u-values of x-line $(n, k-1)$ when computing the system for the u-values for x-line (n, k). In the remainder of this paper, we call this *half-Gauss–Seidel line relaxation.* In principle, grid planes may now be solved concurrently and/or the solution of all tridiagonal systems (n, k) for fixed n may be vectorized. Because the nodes of our target computers contain vector processors, all our implementations use the vectorization in the z-direction for the x-line-relaxation step.

The $M \times N \times K$ grid is distributed over a $P \times Q \times R$ process grid. The data distribution over P processes in the x-direction forces us to use a distributed tridiagonal solver for the x-line-relaxation step; see §3. For Jacobi and half-Gauss-Seidel line relaxation, the data distribution over R processes in the z-direction allows us to solve systems (n, k_0) and (n, k_1) concurrently if they are mapped to different processes. Systems (n, k) with fixed n mapped to the same process may be solved using a solver that is vectorized in the z-direction.

For Jacobi line relaxation, the data distribution over Q processes in the y-direction allows us to solve systems of a grid plane concurrently. For half-Gauss-Seidel line relaxation, however, the tridiagonal systems within one grid plane $k =$ constant still depend on one another, because system (n, k) depends on $(n-1, k)$. This prevents concurrency within one plane. A further modification allows us to obtain a concurrent program: at the process boundaries in the y-direction, we use old u-values for x-line $(n-1, k)$ in the computation of system (n, k). This *modified half-Gauss–Seidel line relaxation* allows concurrency and vectorization for all grid blocks of the $P \times Q \times R$ process grid.

We shall also consider one alternative to line relaxation. *Segment relaxation* avoids distributed tridiagonal systems by moving terms that couple the system across process boundaries to the right-hand side. For an elementary x-line-relaxation step, this leads to systems with coefficient matrices of the form:

$$T' = \left(\begin{array}{cccccccccc}
b_1 & c_1 & & & & & & & & \\
a_2 & b_2 & c_2 & & & & & & & \\
 & a_3 & b_3 & & & & & & & \\
\hline
 & & & b_4 & c_4 & & & & & \\
 & & & a_5 & b_5 & c_5 & & & & \\
 & & & & a_6 & b_6 & & & & \\
\hline
 & & & & & & \cdot & \cdot & & \\
 & & & & & & \cdot & \cdot & \cdot & \\
 & & & & & & & \cdot & \cdot & \\
\hline
 & & & & & & & b_{M-2} & c_{M-2} & \\
 & & & & & & & a_{M-1} & b_{M-1} & c_{M-1} \\
 & & & & & & & & b_M & c_M
\end{array}\right).$$

Such systems can be solved without any communication and only require solving a tridiagonal system in each process. In §4.2.4, we shall see that, for our application, the price for simplicity is a decreased convergence rate and a lack of robustness.

A discussion of point-relaxation methods is omitted, because they diverge when used for our application. We focus the remainder of this paper on modified half-Gauss-Seidel line, Jacobi line, red-black line, and segment relaxation. Each has a certain convergence rate, which is dependent on the particular application. We postpone a discussion of convergence rates until §4, where our application is introduced.

For simplicity of exposition, the relaxation methods were described with the Poisson equation in mind. Their use in our Navier-Stokes application is more complicated, because every grid point corresponds to four unknowns. In (3), the unknowns u_m and the right-hand side coefficients d_m are vectors of dimension 4. The coefficients a_m, b_m, and c_m are 4×4 matrices. The coefficient matrix T of (4) is block-tridiagonal instead of tridiagonal, and its blocks are 4×4 matrices.

3. Distributed block-tridiagonal solvers. In this section, we shall present several methods to solve sets of block-tridiagonal systems (3) defined on an $M \times N \times K$ computational grid that is distributed over a $P \times Q \times R$ process grid. Each solver can be combined with at least one relaxation method of §2.

In process (p, q, r) of the process grid, we have available an $I \times J \times L$ subgrid of the computational grid. The solution of the discretized Navier-Stokes equations requires alternating-direction line relaxations. Hence, even if the grid is distributed in one dimension only (say $Q = R = 1$ or $P = Q = 1$), process boundaries cross the relaxation lines in at least one direction. We consider only the x-line relaxation with systems of M equations distributed over P processes; the extension to y- and z-line relaxations and to nondistributed line relaxations are obvious.

We shall introduce six different distributed tridiagonal solvers. Each can be converted into a block-tridiagonal solver that can be used to solve the systems that arise in our Navier-Stokes application. To convert, scalars of the tridiagonal solvers must be changed into 4×4 matrices or vectors of dimension 4. Divisions like

$$u_m = \frac{a_m}{b_m}$$

must be replaced by 4×4 systems of equations:

$$b_m u_m = a_m.$$

For the remainder of this section, we treat all coefficients and unknowns as scalars and all line-relaxation systems as tridiagonal. The conversion to block-tridiagonal systems considerably complicates the implementation, but it does not affect the fundamental algorithmic structure.

3.1. Pipelining. The classical sequential direct solver for tridiagonal systems, sometimes referred to as the Thomas algorithm, can be used on distributed data. Although a sequential algorithm, there is sufficient concurrency left, because we must solve many tridiagonal systems. For all practical purposes, this solver can only be used in conjunction with the Jacobi line-relaxation method, for which all tridiagonal systems are independent of one another. In the y- and z-directions, the Jacobi x-line relaxation is concurrent, and the only communication requirement is a boundary exchange.

An elementary x-line-relaxation step uses all processes (p, q, r) with $0 \leq p < P$ and q and r fixed. We now use the J systems of the y-direction to introduce concurrency in the x-direction by a *pipelining* technique (this technique is also studied by Ho and Johnsson [8]).

Assuming that the coefficient matrix T is factored once, the Thomas algorithm solves tridiagonal systems by means of two elementary sequential loops. After data distribution over P processes, these loops remain sequential. In process (p, q, r), we must run through J identical loops. As soon as process (p, q, r) hands over the loop of system j to process $(p + 1, q, r)$, process (p, q, r) may start evaluating the loop of system $j + 1$. Filling the pipeline requires P elementary x-line-relaxation steps.

This procedure is also vectorized in the z-direction: instead of evaluating the loop for one system, all L loops of the z-direction are evaluated. This also reduces communication latency, because messages of L systems are combined into one message. On the other hand, this merging of communication increases the time required for filling the pipeline.

3.2. Concurrent direct tridiagonal solvers. As an alternative to pipelining, it is possible to replace the sequential tridiagonal solver by a concurrent direct tridiagonal solver. This avoids the cost of pipelining. However, all known concurrent direct tridiagonal solvers carry a high floating-point overhead. We have used four different direct solvers. From a fundamental point of view, all four are equivalent. They solve a tridiagonal system by LU-decomposition without pivoting, and their operation counts are very similar. Performance differences are due to technical implementation details.

To identify the different solvers, we follow the nomenclature of Frommer [7] in this paper. Note, however, that there is no generally agreed-upon terminology in the literature. For example, the term "recursive doubling" is used for several methods [7], [15]. We shall use "recursive doubling" and "cyclic reduction" for two variants of the cyclic odd-even reduction algorithm, first proposed by Hockney [9].

3.2.1. Recursive doubling. By eliminating unknowns and equations, one can obtain a new tridiagonal system with half the number of unknowns of the original system. This reduction procedure is repeated until a system is obtained that can be solved immediately.

In the recursive-doubling algorithm, one reduction is computed for each of the M equations. Hence, we can determine all solution components immediately after the last reduction step.

3.2.2. Cyclic reduction. The cyclic-reduction method also performs repeated reductions of the system (3). However, it computes only one solvable equation for the whole system. Subsequently, the solution is computed through back-substitution.

In general, the number of equations per process, I, is greater than one. Therefore, we have to distinguish between a local and a global reduction. In the first $\log_2 I$ steps, we reduce the system such that the unknowns are coupled exclusively with unknowns in neighboring processes. Each tridiagonal system now consists of only one equation per process. In the

$$\left(\begin{array}{ccccccccc}
b_1 & & c_2 & & & & & & \\
& b_2 & c_2 & & & & & & \\
& & b_3 & & & c_3 & & & \\
\hline
& & a_4 & b_4 & & c_4 & & & \\
& & a_5 & & b_5 & c_5 & & & \\
& & a_6 & & & b_6 & & & c_6 \\
\hline
& & & & & a_7 & b_7 & & c_7 \\
& & & & & a_8 & & b_8 & c_8 \\
& & & & & a_9 & & & b_9
\end{array}\right)$$

FIG. 1. *Matrix T after eliminating the upper and lower diagonal entries.*

subsequent global reduction, we eliminate the remaining equations of the first, second, fourth, etc. neighbors.

3.2.3. Partition method. In §3.2.2, the local part of the odd-even reduction reduced the system to a new tridiagonal system with one equation per process. The required communication and the arithmetic overhead of solving the latter system are high. This problem is avoided in Wang's partition method [16].

After an elimination of the upper and lower diagonal entries, the coefficient matrix has the shape given in Fig. 1 for a system with $M = 9$ equations distributed over $P = 3$ processes. The fill that results from this operation can be stored in the original arrays a_m and c_m. This matrix is triangularized and, subsequently, diagonalized. For these operations, Wang transposes T from a row-distributed into a column-distributed matrix. Because this transposition of the matrix requires extensive communication, we follow an alternative procedure. We solve the following tridiagonal system with one equation per process:

$$\left(\begin{array}{ccc} b_3 & c_3 & \\ a_6 & b_6 & c_6 \\ & a_9 & b_9 \end{array}\right)\left(\begin{array}{c} u_3 \\ u_6 \\ u_9 \end{array}\right) = \left(\begin{array}{c} d_3 \\ d_6 \\ d_9 \end{array}\right), \tag{5}$$

which is obtained by collecting the equations of the last row in each process. We solved the system (5) by recursive doubling, which we found to be most efficient in the case of one equation per process.

3.2.4. Divide and conquer. In his divide-and-conquer method, Bondeli [2] obtains the solution of (3) by solving a local tridiagonal system in each process concurrently and a global tridiagonal system with one equation in the boundary processes and two equations in the inner processes.

Here, the global problem is solved by eliminating one equation in the inner processes. Again, we obtain a tridiagonal system with one equation per process which is solved with recursive doubling.

3.3. Concurrent iterative tridiagonal solver. The direct solvers of §3.2 solve the system (3) to round-off error (if the systems are well conditioned). Here, we propose a concurrent iterative solver to compute the solution to a certain tolerance.

Splitting the matrix T into a sum $G + H$, as indicated in Fig. 2, the matrix G contains globally uncoupled tridiagonal blocks, and H contains the coefficients that couple the systems across the process boundaries.

This splitting of the coefficient matrix defines the iteration

$$G\,\vec{u}^{(\mu)} = \vec{d} - H\,\vec{u}^{(\mu-1)}. \tag{6}$$

A multicomputer implementation of (6) is easily obtained and is efficient for several reasons:

$$T = \left(\begin{array}{ccc|ccc|ccc} b_1 & c_1 & & & & & & & \\ a_2 & b_2 & c_2 & & & & & & \\ & a_3 & b_3 & & & & & & \\ \hline & & & b_4 & c_4 & & & & \\ & & & a_5 & b_5 & c_5 & & & \\ & & & & a_6 & b_6 & & & \\ \hline & & & & & & b_7 & c_7 & \\ & & & & & & a_8 & b_8 & c_8 \\ & & & & & & & a_9 & b_9 \end{array}\right) + \left(\begin{array}{ccc|ccc|ccc} & & & & c_3 & & & & \\ \hline & & & a_4 & & & & & \\ & & & & & & & & c_6 \\ \hline & & & & & & & a_7 & \\ & & & & & & & & \end{array}\right)$$

FIG. 2. *Splitting of T for M = 9 and P = 3.*

1. Because G contains only the uncoupled tridiagonal matrices, its inversion is trivially concurrent. The LU-decomposition of G needs to be carried out only once. We can solve (6) by computing the right-hand side terms and by evaluating two back-substitution loops in each process.

2. The matrix-vector operation $\vec{d} - H\,\vec{u}^{(\mu-1)}$ requires only nearest-neighbor communication.

3. Only the right-hand side terms of the first and the last row in each process change within one iteration step. Hence, the arithmetic costs of one iteration step are low.

4. A good initial guess for the iteration is found by solving the system

$$G\,\vec{u}^{(0)} = \vec{d}. \tag{7}$$

5. The iteration (6) is continued until some stopping criterion is satisfied. This criterion can be chosen to fit the need and, usually, depends on the specific application; see §4.2.5 for a further discussion.

4. Computational results. In this section, we compare the performance on the Intel Touchstone Delta of the relaxation methods of §2 implemented using the block-tridiagonal solvers of §3. All performance data are obtained for a solver of the Navier–Stokes equations for three-dimensional, incompressible, unsteady, and viscous flows. In §4.1, we describe the problem in sufficient detail to understand the computational complexity of the application. However, fluid-dynamical results obtained with this code are published elsewhere [1]. Additional physical and numerical details are found in [3] and [4].

Our problem sizes and convergence properties are not artificial creations. The problem size was determined by realistic resolution requirements and was NOT increased to obtain artificially high efficiency typically associated with coarse-grain computations. Similarly, our selection of numerical methods, i.e., of line-relaxation methods over point-relaxation methods, was guided by realistic needs. There is no compelling reason to use line-relaxation methods for the Poisson equation, because one can always use simple point-relaxation methods. This alternative is not available now, because point-relaxation methods do not even converge for our application. In §4.2.4, we shall also observe that segment relaxation is not sufficiently robust for our Navier–Stokes problem. It is very important that our performance measurements are obtained for a "real" problem. The grid size and the accuracy with which the discrete problems are solved are dictated by the physics of our application.

In our performance analysis of concurrent relaxation methods, we distinguish between numerical and implementation issues. Numerical issues primarily determine the number of relaxation-iteration steps necessary for convergence. This is, of course, extremely application dependent. Implementation issues primarily determine the execution time of one relaxation step through cost of communication, load balance, number of processes, etc. This depends

on the application through computational parameters like grid dimensions and number of unknowns. Nevertheless, the performance results of one relaxation step are more readily transferred to other applications. For this reason, we consider these two performance issues separately.

We shall present the convergence rates for the different relaxation methods in §4.2. Multicomputer performance of one alternating-direction line-relaxation step will be discussed in §4.3. Convergence-rate and per-step-performance information are combined into global-performance results in §4.4.

4.1. Discretization of the Navier–Stokes equations. The Navier-Stokes equations are a set of coupled nonlinear partial-differential equations, which describe the conservation of mass, momentum, and energy for continuous media. For an incompressible fluid, a time-dependent flow in three dimensions is defined by the pressure $p(x, y, z, t)$ and the three velocity components $u(x, y, z, t)$, $v(x, y, z, t)$, and $w(x, y, z, t)$. In a dimensionless matrix-vector form, the conservation equations are given by

$$\overline{R} \cdot \frac{\partial Q}{\partial t} + \frac{\partial F}{\partial x} + \frac{\partial G}{\partial y} + \frac{\partial H}{\partial z} = \frac{1}{Re} \cdot \left(\frac{\partial^2 S}{\partial x^2} + \frac{\partial^2 S}{\partial y^2} + \frac{\partial^2 S}{\partial z^2} \right), \tag{8}$$

where

$$\overline{R} = \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}, \quad Q = \begin{pmatrix} p \\ u \\ v \\ w \end{pmatrix}, \quad S = \begin{pmatrix} 0 \\ u \\ v \\ w \end{pmatrix},$$

$$F = \begin{pmatrix} u \\ u^2 + p \\ uv \\ uw \end{pmatrix}, \quad G = \begin{pmatrix} v \\ vu \\ v^2 + p \\ vw \end{pmatrix}, \quad H = \begin{pmatrix} w \\ wu \\ wv \\ w^2 + p \end{pmatrix}.$$

Breuer and Hänel [4] extended the method of artificial compressibility to unsteady flows. They define an artificial time τ and add a supplementary time derivative $\hat{R}\frac{\partial Q}{\partial \tau}$, with

$$\hat{R} = \begin{pmatrix} \frac{1}{\beta^2} & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix},$$

to the Navier-Stokes equations. The pressure field is now coupled to the velocity distribution, and (8) can be integrated. Because a steady solution is computed at time τ, the additional terms vanish, and we obtain the unsteady solution of (8) at the physical time t.

Let ξ be an index for the physical time step and ζ an index for the artificial time step. The approximation

$$\frac{\partial Q}{\partial \tau} \approx \frac{Q^{\xi+1,\zeta+1} - Q^{\xi+1,\zeta}}{\Delta \tau} = \frac{\Delta Q^{\zeta}}{\Delta \tau}$$

leads to an implicit discretization scheme for the artificial time step. In this scheme, the Euler and viscous fluxes are defined at the new physical time step $\xi + 1$ and at the new artificial time step $\zeta + 1$. Consider, e.g., the Euler flux F. The implicit discretization uses its value $F^{\xi+1,\zeta+1}$ at physical time step $\xi + 1$ and artificial time step $\zeta + 1$. This term is expanded in a Taylor series with respect to the artificial time τ to obtain the following linearization:

$$F^{\xi+1,\zeta+1} = F^{\xi+1,\zeta} + J_F^{\xi+1,\zeta} \Delta Q^{\zeta}.$$

Here, J_F is the Jacobian matrix of the Euler flux F with respect to Q, say $J_F = \frac{\partial F}{\partial Q}$. The resulting discrete system of equations is given by

$$\text{LHS} \cdot \Delta Q^{\zeta} = \text{RHS}\,. \tag{9}$$

The right-hand side contains the expressions that arise from the discretization of the derivatives in (8) using a high-order upwind scheme for the convective terms, central differences for the diffusive terms, and a second-order discretization for the physical time t. The left-hand side contains first-order spatial derivatives of the Jacobian matrices.

Because the left-hand side terms in (9) vanish for the exact solution vector Q, the discrete solution has the order of accuracy of the right-hand side. The numerical advantage is that the spatial discretization of the Jacobian matrices has no influence on the order of accuracy of the solution. Therefore, we have gained some flexibility in choosing a discretization of LHS. This flexibility is used to improve the diagonal dominance of the coefficient matrix and to increase the size of the time step.

Although central differences would be the most straightforward choice, they do not add terms to the main diagonal of the coefficient matrix. That is why we use a first-order upwind formulation combined with a matrix-splitting technique: after splitting the Jacobian matrices into positive and negative parts, forward differences are applied to the positive parts and backward differences to the negative parts. Consider, e.g., the Jacobian matrix J_F. This 4×4 matrix is diagonalized such that

$$J_F = M \Lambda M^{-1},$$

where M is the matrix of eigenvectors of J_F, and Λ is the diagonal matrix containing the eigenvalues of J_F on its diagonal. Let

$$\Lambda = \Lambda^{+} + \Lambda^{-},$$

where Λ^{+} and Λ^{-} contain, respectively, the positive and negative eigenvalues. After transforming back to the Jacobian J_F, we obtain the splitting

$$J_F = M \Lambda^{+} M^{-1} + M \Lambda^{-} M^{-1} = J_F^{+} + J_F^{-}.$$

The spatial derivatives in J_F^{+} and J_F^{-} are discretized, respectively, with forward and backward differences. This method tends to increase the diagonal entries of the coefficient matrix, and the resulting system of equations can be solved with a line-relaxation method. For further details and for a validation of the above scheme, refer to Breuer [3] and Breuer and Hänel [4].

Every solution component consists of four elements: the pressure and the three velocity components. Hence, when applying the methods described in §3, the system (3) becomes a block-tridiagonal system, the coefficients a_m, b_m, and c_m are 4×4 matrices, and the right-hand side terms d_m are vectors consisting of four components. It follows that the discretized Navier–Stokes equations require much more arithmetic per grid point than, e.g., the discretized Poisson equation.

All computations of this paper use the above method to simulate the flow of an isolated slender vortex embedded in an axial flow. The scientific interest in this flow is the study of the phenomenon of a bursting vortex or vortex breakdown. Although the initial conditions are axially symmetric, the flow is fully three-dimensional after vortex breakdown. The boundary conditions in the outflow plane and on the lateral boundaries are extrapolated from the case of a free vortex. At the outflow plane, a nonreflecting boundary condition simulates undisturbed vortical flow through the boundary. At the lateral boundaries, the pressure is imposed such

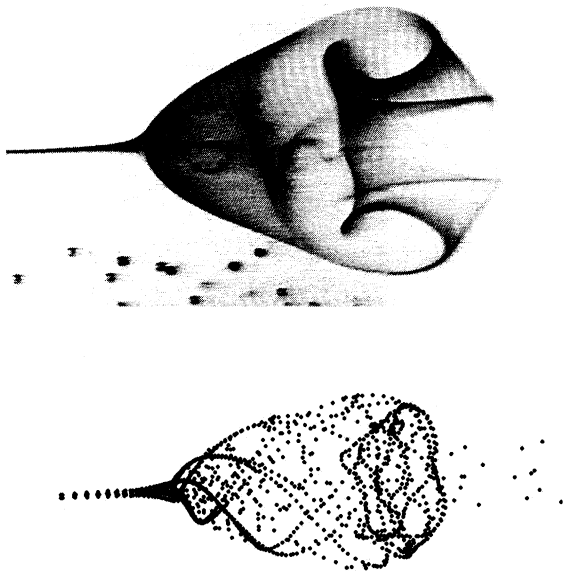

FIG. 3. *Experimental and numerical flow visualization of bubble-type vortex breakdown: top picture shows experimental streaklines and bottom picture shows numerical streaklines.*

that the bursted part of the vortex lies well within the domain of integration. Breuer [3] gives complete details on the initial and on the boundary conditions required to simulate this flow.

Figure 3 compares a numerical simulation and an experiment of Althaus et al. [1]. Using a Reynolds number based on the radius of the vortex core as length scale and the mean axial velocity as velocity scale, both cases have a Reynolds number of 500. Our simulation uses a $32 \times 32 \times 64$ rectangular grid with the z-direction being the axial coordinate. The physical time step is 0.3, and the artificial time step is 100. Althaus et al. [1] perform a detailed comparison of numerical and experimental results. It is this computation that we use to evaluate the performance of all our codes.

Depending on the orientation of the relaxation line, we have to solve $32 \times 64 = 2048$ systems of 32 blocks or $32 \times 32 = 1024$ systems of 64 blocks of size 4×4. Our challenge is to make effective use of all 512 computing nodes of the Delta to solve the block-tridiagonal systems of this moderate size. As discussed in §2, we use a $P \times Q \times R$ process grid. Preliminary tests, which we do not report, showed that three-dimensional grid distributions outperform one- and two-dimensional grid distributions with $P = 1$ and/or $Q = 1$, in spite of the fact that the latter can use efficient sequential tridiagonal solvers in at least one direction. This may seem somewhat counterintuitive. Given a certain number of nodes, choosing $P = 1$ leads to higher values for Q and/or R. The fact that x-line-relaxation steps are very efficient when $P = 1$ is offset by the increased load imbalance and the decreased efficiency in the other directions. Therefore, we always use a process grid with $P \approx Q \approx R$; see Table 1. In all our computations, each process is mapped to a separate node of the multicomputer.

The experiments of this section were performed on the Intel Touchstone Delta [5], which consists of an ensemble of 512 computing nodes connected in a two-dimensional mesh. The computational engine of each node is an Intel i860 processor with an advertised peak perfor-

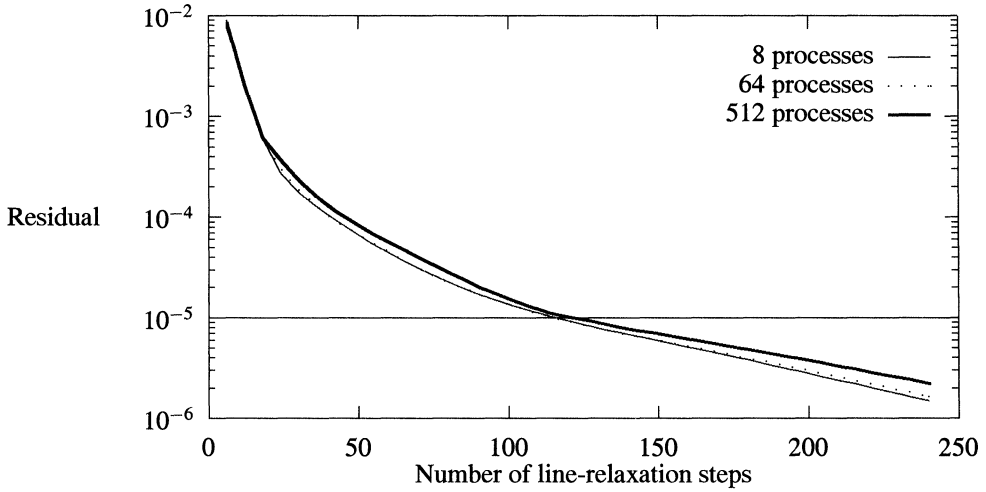

FIG. 4. *Maximal residual as a function of the number of modified half-Gauss–Seidel line-relaxation steps for three different process grids.*

TABLE 1
Process grids used in our computational tests.

Total Number	Number of processes in x-direction (P)	y-direction (Q)	z-direction (R)
4	1	1	4
8	2	2	2
16	2	2	4
32	2	2	8
64	4	4	4
128	4	4	8
256	4	4	16
512	8	8	8

mance of 60 double-precision MFLOPS. Each node has a local memory of 16 MB. On the Delta, our standard problem needs a minimum of four nodes because of memory requirements.

4.2. Convergence of relaxation methods.

4.2.1. Modified half-Gauss–Seidel line relaxation. The modified half-Gauss-Seidel line relaxation uses the updated unknowns of the previously computed plane except at process boundaries; see §2. The convergence rate of the modified method is dependent on the number of processes and on the choice of process boundaries, because they determine the matrix splitting underlying the relaxation method. Figure 4 shows the residual as a function of the number of relaxations for some of the partitions of Table 1. Although the convergence depends on the partition, all computations achieve the required residual of 10^{-5} within approximately the same number of iteration steps.

4.2.2. Jacobi line relaxation. In the case of Jacobi line relaxation, all relaxation lines may be computed simultaneously. The convergence rate of the Jacobi line relaxation does not depend on the number of processes. Note that the modified half-Gauss-Seidel line relaxation turns into the Jacobi line relaxation when the number of processes is equal to the number of grid planes.

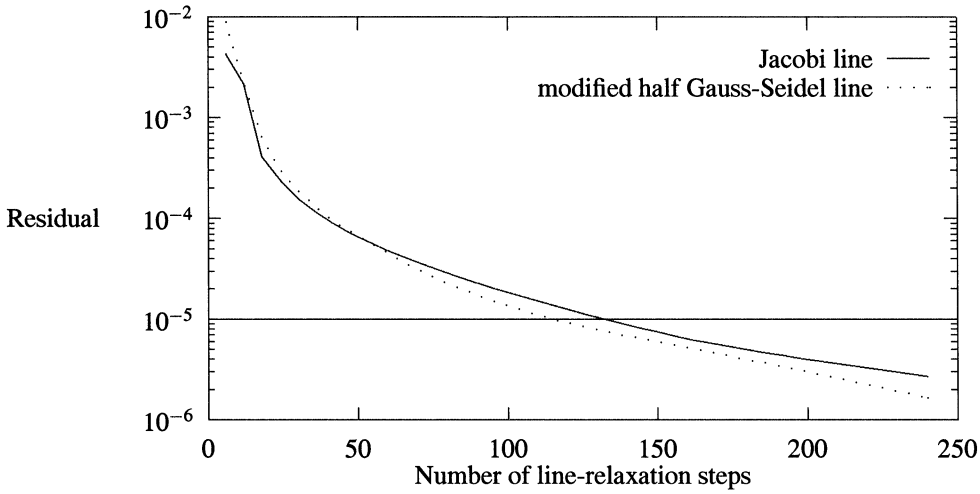

FIG. 5. *Maximal residual as a function of the number of line-relaxation steps in a computation with* 64 *processes.*

In Fig. 5, half-Gauss-Seidel and Jacobi line relaxation on the 64-node partition are compared. The Jacobi method needs about 10% more iteration steps to reach the required tolerance on the residual.

4.2.3. Red-Black line relaxation. Like the Jacobi line relaxation, the convergence rate of the Gauss-Seidel line relaxation with red-black ordering does not depend on the number of processes. We obtained a test program for the red-black line relaxation easily by splitting the Jacobi relaxation into the two steps explained in §2 and changing the step width of the inner loops to two. After the first step, the updated unknowns have to be exchanged between the processes. Besides the required additional communication, there is a loss of performance, because the inner loops can no longer be vectorized.

For our application, there is no observable difference in the convergence rate between the red-black and the Jacobi line iteration. The red-black line relaxation is, therefore, not a viable alternative to the modified half-Gauss-Seidel line method and, because of the communication and vectorization losses, its global performance is worse than the Jacobi line relaxation.

4.2.4. Segment relaxation. The segment relaxation computes relaxation lines interrupted by the process boundaries. Efficiency and applicability of this method depend on the convergence rate of the resulting iteration. Consider a segment-relaxation step in one direction with M grid points distributed over P processes, and let

$$I = \frac{M}{P}$$

be the number of local grid points. For $I = M$, we obtain a sequential line relaxation, while a Jacobi point relaxation results if $I = 1$.

We implemented an alternating-direction segment relaxation and plot the residual as a function of the number of relaxation steps for a 256- and a 512-node partition in Fig. 6. The plot shows that the relaxation fails to converge for the unbalanced $4 \times 4 \times 16$ partition. The $8 \times 8 \times 8$ partition needs about 20% more iteration steps than the modified half-Gauss-Seidel line relaxation.

We observed a nonconverging iteration even for the $2 \times 2 \times 8$ partition. We conclude that segment relaxation is not suitable, because robustness of the algorithm cannot be guaranteed.

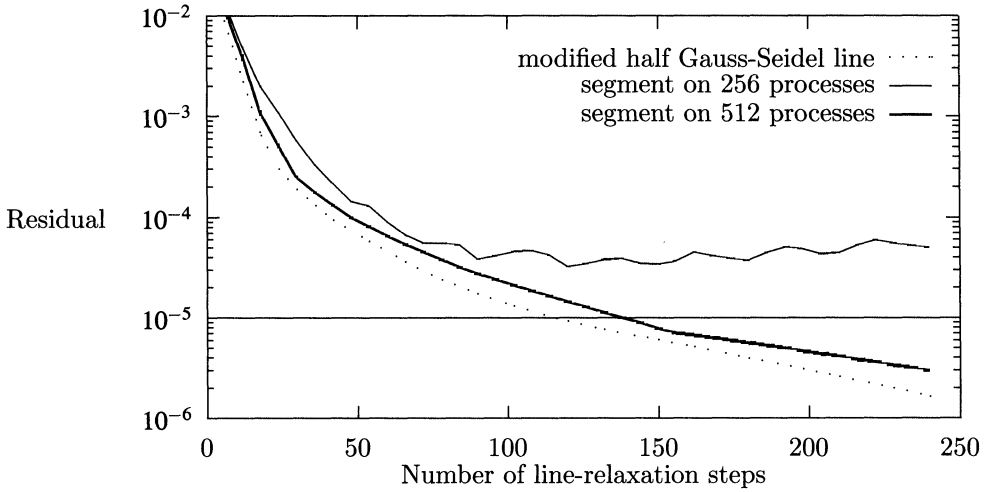

FIG. 6. *Maximal residual as a function of the number of line-relaxation steps for segment relaxation and modified half-Gauss–Seidel line relaxation.*

Although we achieve convergence when using all nodes on the Delta, it is uncertain whether the code would converge on a larger configuration of the same hardware. We consider this nonrobust behavior unacceptable.

4.2.5. Iterative block-tridiagonal solver. The modified half-Gauss-Seidel line relaxation of §4.2.1 can also be implemented using the concurrent iterative block-tridiagonal solver of §3.3. Because this method solves the system (3) only approximately, one should consider the effect of less-accurate block-tridiagonal solvers on the convergence rate of the line-relaxation method.

In Fig. 7, we examine the impact on the number of line-relaxation steps of using the iterative solver with six block-tridiagonal iteration steps. (The rationale for using six iteration steps is discussed in §4.3.6.) The curves for the exact and iterative solvers show the same progress, and the required residual of 10^{-5} is reached almost within the same relaxation step.

With an increasing number of processes, the initial guess (7) gets worse, and more coupling coefficients are set equal to zero to obtain the tridiagonal-iteration matrix. The partition, therefore, has an impact on the convergence rate. Figure 8 compares three different partitions. Up to 256 processes, the curves virtually coincide. The computation with 512 nodes requires only a few additional line-relaxation steps to converge.

4.3. Performance of one alternating-direction line-relaxation step. In this section, we shall compare the multicomputer performance of one alternating-direction line-relaxation step, i.e., one line relaxation in each spatial direction. To obtain a dimensionless efficiency for a multicomputer program, one must relate the execution time to a meaningful reference time. In principle, this reference time must be the best execution time possible on one node of the same multicomputer. In reality, one must settle for the sequential-execution time of a reasonable but not necessarily the best sequential program. For us, even the reasonable sequential time is impossible, because our problem requires too much memory to solve on one node. We defined a fictitious reference time in the following manner. To obtain a sequential time for the relaxation in the x-direction, we distribute the grid only in the y- and z-directions over the processes. Subsequently, the concurrent-execution time for this partition is then multiplied by the number of processes. We repeat this method for the y- and z-directions. Our

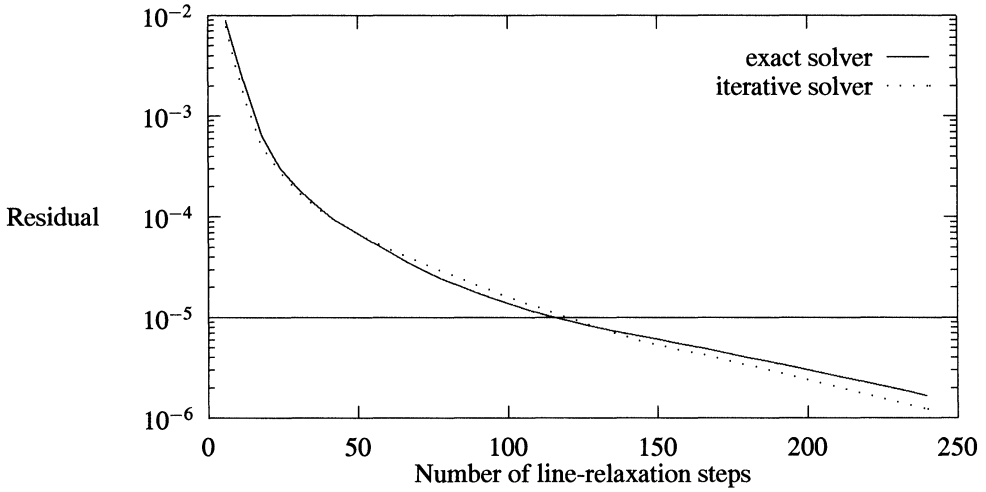

FIG. 7. *Maximal residual as a function of the number of line-relaxation steps when using the exact and the iterative block-tridiagonal solver to implement modified half-Gauss–Seidel line relaxation with* 64 *processes.*

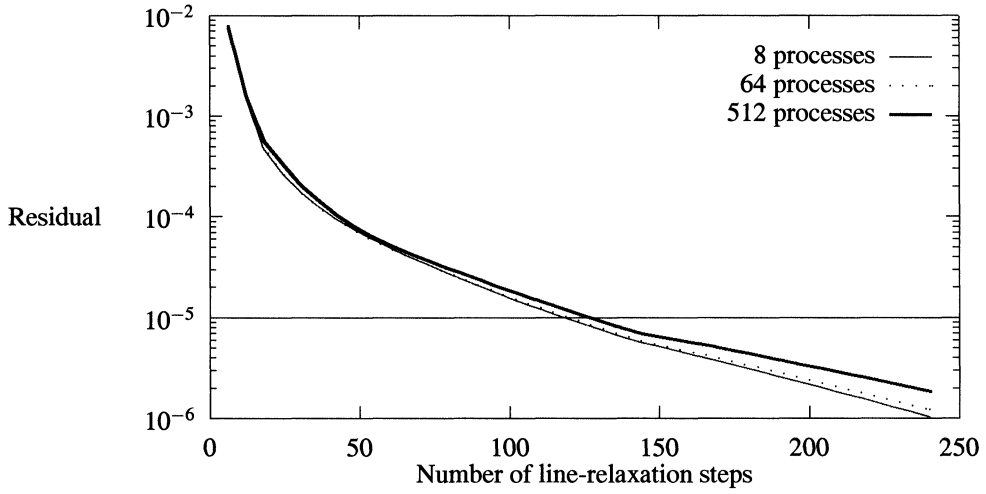

FIG. 8. *Maximal residual as a function of the number of line-relaxation steps to examine impact of process grids on the iterative block-tridiagonal solver.*

sequential-execution time for the alternating-direction line relaxation is the sum of these three terms. With process boundaries that are parallel to the relaxation lines, the block-tridiagonal systems can be solved by sequential LU-decomposition, which is the optimal procedure for sequential computations.

The same reference time is used for all methods, such that the efficiency results can be compared directly. The efficiency η_P of a P-node computation is the ratio

$$\eta_P = \frac{\text{(reference sequential time)}}{P \cdot \text{(execution time of the } P\text{-node program)}}.$$

We shall compare the multicomputer performance by means of two different kinds of plots. The first kind plots the efficiency η_P as a function of the number of nodes, and the second

plots the execution time T_P in seconds as a function of the number of nodes. In both plots, a logarithmic scale is used for the number of nodes. In the execution-time plot, we also use a logarithmic scale for the second axis such that the line of linear speedup has slope -1. (The slope is distorted, however, because different scaling factors are used for the horizontal and vertical axes.)

The increased computational requirements of the discrete Navier–Stokes equations over, e.g., the discrete Poisson equation actually results in a more efficient computation, because more arithmetic occurs for the same number of messages. Although the messages are longer for the discrete Navier–Stokes equations, the latency usually dominates the communication time, and the length of the messages is less important than the number of messages.

4.3.1. Pipelining. In §3.1, we described a pipelining method for the sequential LU-decomposition. This makes sense if the number of processes is small compared with the number of unknowns per block-tridiagonal system [8]. This method, compared with all other methods we implemented, has the lowest floating-point operation count and requires the least amount of communication. However, it is less concurrent than some other methods because of load imbalance. The pipelining implements the Jacobi line relaxation.

Figure 9 shows the performance of the pipelining algorithm. As expected, the algorithm shows good efficiency for a small number of processes. However, for larger numbers of processes, the startup time required for all processes to participate in the computation causes a progressive loss of efficiency.

4.3.2. Recursive doubling. A concurrent implementation of the modified half-Gauss-Seidel line relaxation with the recursive-doubling algorithm for block-tridiagonal systems requires more arithmetic and more communication than the pipelining method, but offers higher concurrency. The recursive-doubling computations in the execution-time plot of Fig. 10 lie on a line that is almost parallel to the line of linear speedup. These lines are parallel and the efficiency is constant as a function of the number of nodes, because the communication overhead plays only a secondary role in computations with up to 512 nodes. The vertical distance between these two lines can be related to the arithmetic overhead, which is mainly responsible for the disappointing efficiency of about 15%.

4.3.3. Cyclic reduction. The cyclic-reduction algorithm is based on the same reduction procedure as recursive doubling and is also an implementation of the half-Gauss–Seidel line relaxation. This method needs less arithmetic, but additional communication for the back-substitution. Whereas the local part of the cyclic reduction shows a good load balance, the number of processes that participate during the global part halves after each reduction step. Both facts lead to a significant loss of efficiency for large numbers of processes (Fig. 10). However, cyclic reduction always beats recursive doubling. It beats the pipelining method for computations with all available nodes (see also Table 2).

For the back-substitution, the cyclic-reduction algorithm must store the entries of the coefficient matrix T and the right-hand side terms in (3), which change after each reduction step. The additional memory made a run on four nodes impossible.

The dips in the efficiency curves for 32 and 256 processes occur, because we double the partitions in the x- and y-directions (see Table 1). In these cases, the z-line-relaxation steps are inefficient compared to the next larger partition ($2 \times 2 \times 8 \rightarrow 4 \times 4 \times 4$ and $4 \times 4 \times 16 \rightarrow 8 \times 8 \times 8$, respectively).

4.3.4. Partition method. When modified half-Gauss–Seidel line relaxation is implemented by means of Wang's partition method, we realize an improved efficiency compared with the cyclic-reduction algorithm; compare Figs. 10 and 11. We examined in detail execution profiles of the z-line relaxation with a $1 \times 1 \times 8$ process grid. In this case, already

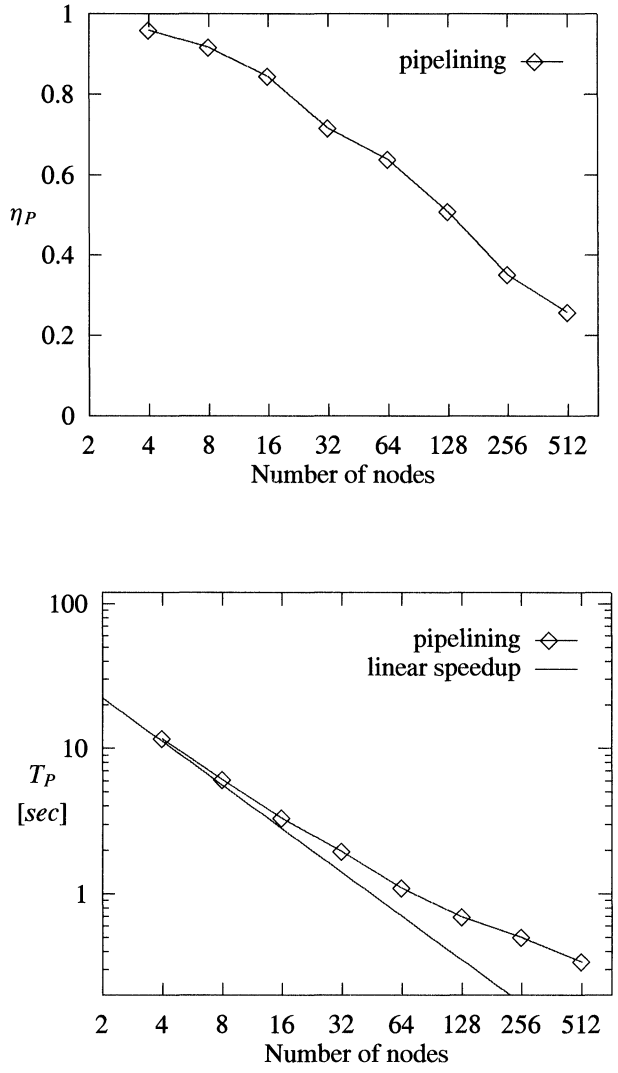

FIG. 9. *Efficiency and execution time of the pipelining algorithm.*

about 40% of the execution time was needed for solving the reduced system computed in the first part of Wang's partition method. This step is communication intensive and becomes even more so when the number of processes is increased further. It is this part of the computation that is responsible for the gradual loss of efficiency as the number of processes is increased.

4.3.5. Divide and conquer. The performance of the divide-and-conquer algorithm applied to the modified half-Gauss-Seidel line-relaxation method is given in Fig. 11. Like the partition method, divide and conquer reduces the size of the tridiagonal systems down to one block of equations per process.

The communication needed for the solution of the reduced system, obtained again with recursive doubling, decreases the efficiency of larger partitions in our application. The performance as a function of the number of processes is almost identical to that of the partition

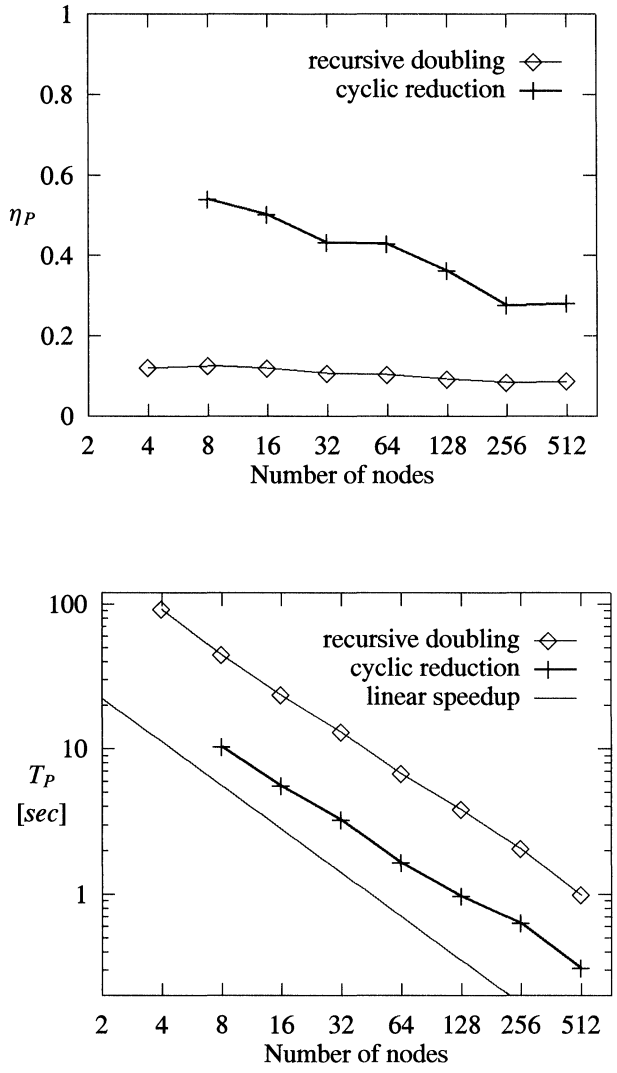

FIG. 10. *Efficiency and execution time of recursive doubling and cyclic reduction.*

TABLE 2
Execution times until convergence on 512 *nodes.*

Block-tridiagonal solver	Execution time in seconds
Iteration method	10.12
Partition method	11.08
Divide and conquer	12.31
Cyclic reduction	12.44
Pipelining	14.96
Recursive doubling	39.64

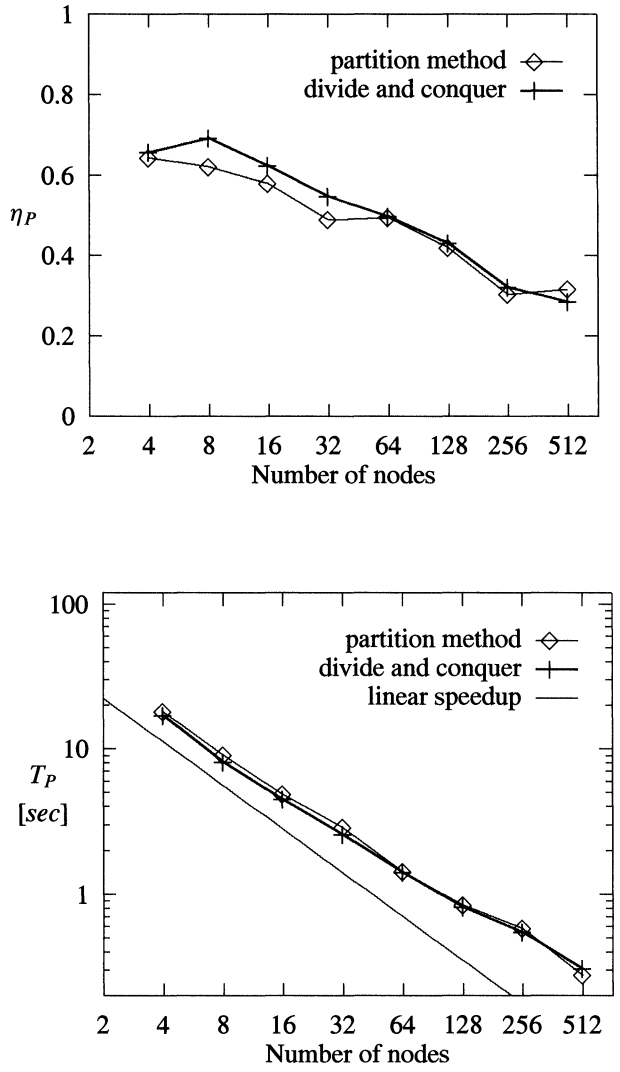

FIG. 11. *Efficiency and execution time of the partition and the divide-and-conquer method.*

method. The divide-and-conquer algorithm outperforms the partitioning method on smaller process grids.

4.3.6. Iterative block-tridiagonal solver. With the iterative block-tridiagonal solver, we implemented the modified half-Gauss-Seidel line relaxation. Because the iteration method solves the block-tridiagonal systems approximately, the accuracy requirements are a decisive determinant for the performance of the iteration method. In principle, it would be possible to iterate until a certain criterion is satisfied, e.g.,

$$|\vec{u}^{(\mu)} - \vec{u}^{(\mu-1)}| \leq \varepsilon, \tag{10}$$

where ε is a certain tolerance. If ε is small, the iterative solver is numerically equivalent to a direct solver. However, if ε is large, the error on the solution of the tridiagonal system may

have an impact on the convergence rate of the line relaxation. The size of ε in (10), therefore, not only determines the number of block-tridiagonal iteration steps, but also the number of line-relaxation steps.

In a multicomputer computation, error-adaptive strategies like those suggested by (10) add significant costs that are difficult to recoup by the expected reduction in the number of iteration steps: computing error estimates is expensive, because they require global communication. Therefore, (10) is not used in our final computation. Instead, we are using a fixed number of block-tridiagonal iteration steps. We found that this is more efficient, while highly reliable if the number of block-tridiagonal iteration steps is sufficiently large. To determine an appropriate number, we performed a sequence of experiments on a $1 \times 1 \times 8$ process grid. In (10), we set $\varepsilon = 10^{-6}$, which is one order of magnitude below the requested residual of the global relaxation. We found that six iterations were always sufficient to reach the required accuracy if the initial guess $\vec{u}^{(0)}$ of (7) was used. The performance results shown in Fig. 12 were all measured with the number of block-tridiagonal iteration steps set equal to 6. With more than 32 processes, the algorithm is more efficient than the partition method and more efficient than the divide-and-conquer technique. With 256 processes, the performance is better than any other concurrent algorithm we implemented.

It was already shown in §4.2.5 that, with six block-tridiagonal iteration steps, there was virtually no difference in the number of line-relaxation steps between the iterative and the exact block-tridiagonal solver. Line-relaxation steps based on either solver are, therefore, practically equivalent. It is possible to reduce the number of block-tridiagonal iteration steps below six. However, the resulting line-relaxation method is no longer equivalent with the original method and may require a larger number of line-relaxation steps. On the other hand, each line-relaxation step may be considerably less expensive. We did not pursue this possibility of trading off the accuracy of the block-tridiagonal solver with the convergence rate of the line-relaxation method. However, segment relaxation is equivalent to line relaxation with one iteration step of the iterative block-tridiagonal solver. Considering the robustness problems of segment relaxation, at least two block-tridiagonal iteration steps are needed.

4.4. Global performance. The interesting performance of a program is, of course, the execution time until convergence within tolerance. In Fig. 13, we compare this time for the Jacobi line-relaxation method implemented with pipelining and for the modified half-Gauss-Seidel line relaxation implemented with the partition method and with the iterative block-tridiagonal solver. Here, we focus only on the execution time needed for the relaxation routines. To obtain the actual time to solve the Navier-Stokes equations, we must add to this the time for the computation of the right-hand side terms and the boundary conditions, which are independent of the solution procedure.

The better performance of the iterative block-tridiagonal solver for computations with more than 256 processes is partially offset by the required additional relaxation-iteration steps; see §4.2.5. The convergence losses of the Jacobi line relaxation (§4.2.2) deteriorate the performance of the pipelining algorithm. The execution times until convergence on 512 nodes for all implementations are given in Table 2. The modified half-Gauss-Seidel line relaxation implemented with the iterative block-tridiagonal solver clearly beats all competitors. As pointed out before, there are several techniques that could improve its performance even further. Most importantly, the fastest method is also, by far, the easiest to implement.

5. Computer dependence. The execution time of a program is, by its very nature, a computer-dependent characteristic. One must, therefore, always question whether or not particular performance results for a set of algorithms obtained on one computer carry over to other computers. Although not our main concern here, comparative performance studies can also be used for computer-evaluation purposes. In a preliminary comparative study, we obtained

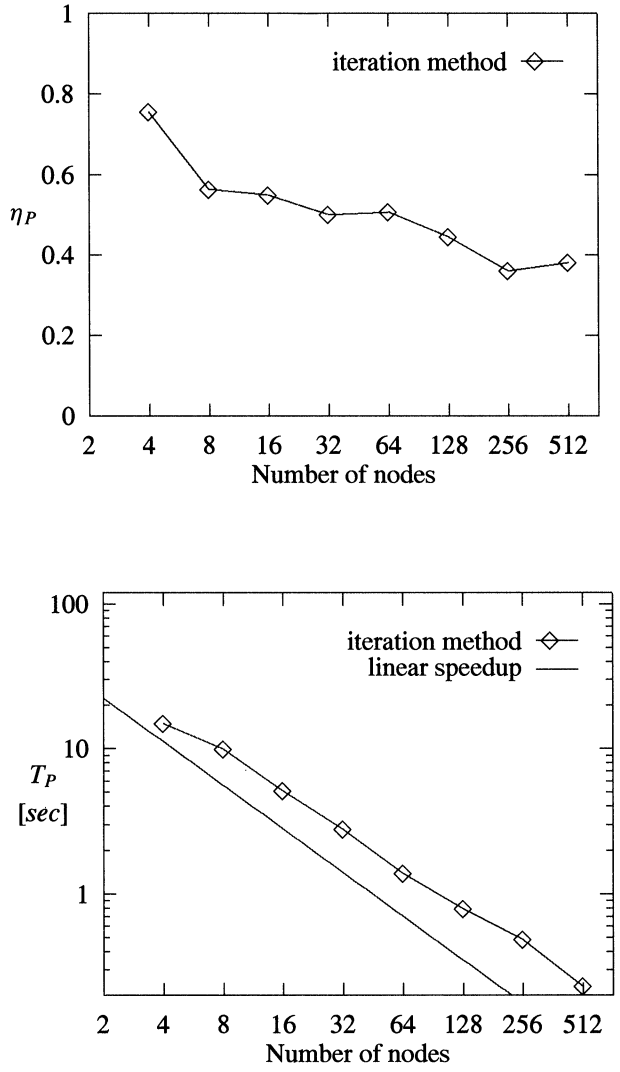

FIG. 12. *Efficiency and execution time of the iteration method.*

some performance data on the Intel Paragon XP/S [6] and the Parsytec SuperCluster [13]. We also used the Fujitsu S-600, which is a conventional vector processor with an advertised peak performance of five GFLOPS.

The Paragon design is similar to that of the Delta. The i860 processors of the Paragon have a higher clock frequency than those of the Delta: 50 MHz instead of 40 MHz. In principle, this increases the advertised peak performance by 25% to 75 MFLOPS per node. Furthermore, Paragon nodes have larger caches and contain improved communication hardware to reduce latency and increase bandwidth.

The Parsytec SuperCluster consists of 256 Transputers T-805, which are clocked at 30 MHz. Each Transputer has an advertised peak performance of 2.2 MFlops. With 4 MBytes of memory per node, our application requires at least 16 Parsytec nodes. Each node contains four bidirectional communication links, which can be used to configure the nodes into a large

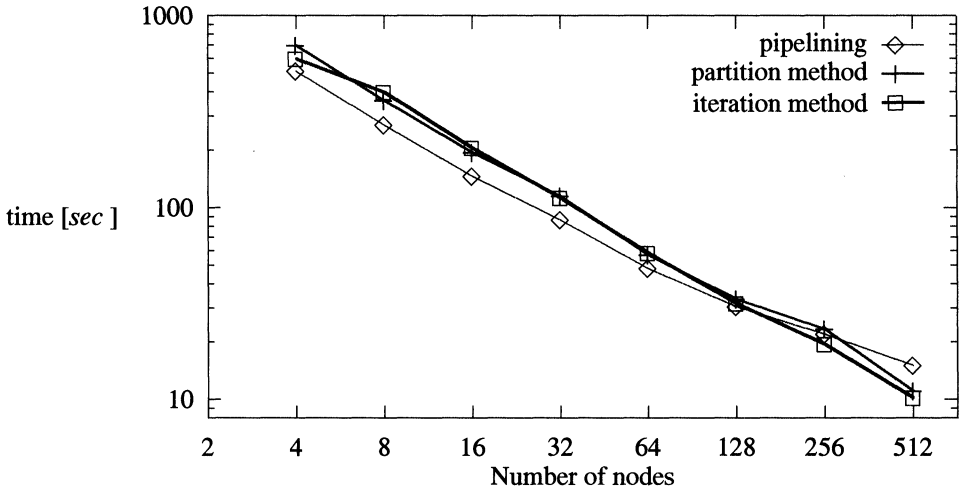

FIG. 13. *Execution time until residual* $\leq 10^{-5}$.

variety of network topologies. The floating-point performance of the Transputer is considerably less than that of a Paragon or a Delta node. However, the Parsytec has an excellent ratio of communication versus arithmetic time.

In Figs. 14, 15, and 16, the execution time of one alternating-direction line-relaxation step is displayed as a function of the number of nodes. Figure 14 considers Jacobi line relaxation implemented using the pipelined block-tridiagonal solver on the Delta, Paragon, and Parsytec. Figures 15 and 16 display execution times obtained with modified half-Gauss-Seidel line relaxation. Figure 15 is for the program based on the concurrent iterative block-tridiagonal solver, and Figure 16 for the program that uses the partition method to solve the block-tridiagonal systems concurrently. In these three figures, the lines of linear speedup are based on a sequential-execution time obtained for the Delta as explained in §4.3. The lines of linear speedup for Paragon and Parsytec are parallel to this line, but are not displayed to avoid crowding the figures; Parsytec efficiency is compared with Delta efficiency in Fig. 17.

In the execution-time plots, the Parsytec computations lie on a line that is almost parallel to the line of linear speedup. This is most pronounced for the method based on the iterative block-tridiagonal solver. This is an indication that almost all overhead on the Parsytec is due to the increased operation count of the concurrent computations and not to communication. This is confirmed in the efficiency plots of Fig. 17, which displays the efficiencies as a function of the number of nodes. (Efficiencies are computed using the sequential-execution time on the computer of the concurrent computation.) The iteration method on the Parsytec has a constant efficiency of about 40%, which indicates that communication overhead is negligible. When executed on one node, the concurrent Parsytec program is about $\frac{1}{0.4} = 2.5$ times slower than the sequential Parsytec program. However, the concurrent program speeds up linearly.

The pipelining method does not have a constant efficiency on the Parsytec. However, its decrease in efficiency is much less pronounced than on the Delta: communication and load-imbalance effects are not as important on the Parsytec as on the Delta. This is, of course, easily explained by the smaller communication-arithmetic ratio of the Parsytec.

Because of the low floating-point-operation count of the pipelining method and the high communication efficiency of the Parsytec, pipelining remains the fastest algorithm for up to the maximum available number of nodes (256). However, extrapolating Fig. 17, we expect that

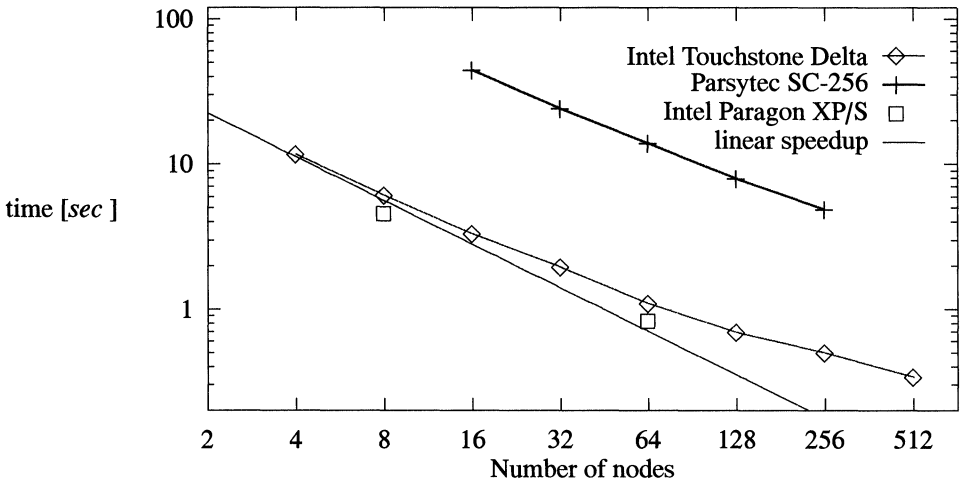

FIG. 14. *Execution time of the pipelining method.*

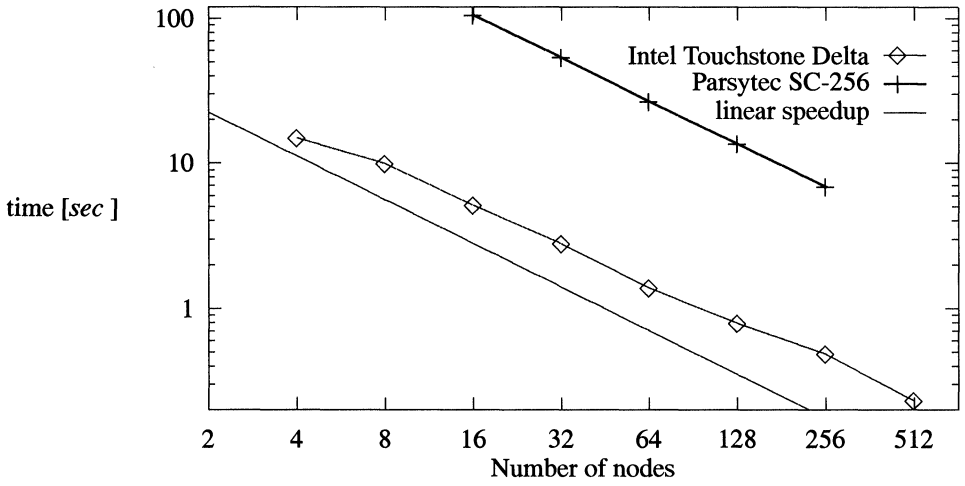

FIG. 15. *Execution time of the iteration method.*

the iteration method will perform better than the pipelining method on Parsytec-like computers with more than 512 nodes.

For our computations, the Delta is about 10–13 times faster than the Parsytec. Based on the advertised peak performance of their nodes, the Delta should be about 30 times faster than the Parsytec. The speed difference between Parsytec and Delta is smaller for fine-grain computations, because the Parsytec is a more efficient computer. The higher efficiency is not enough, however, to close the speed gap with the Delta.

As mentioned before, our code solves the Navier-Stokes equations to simulate time-dependent, incompressible, and unsteady flows. Table 3 displays typical execution times to obtain useful fluid-dynamical results. These require about 1000 physical time steps, and each time step requires many line-relaxation steps. These execution times also include all other computations that surround the actual alternating-direction line-relaxation iteration. The most

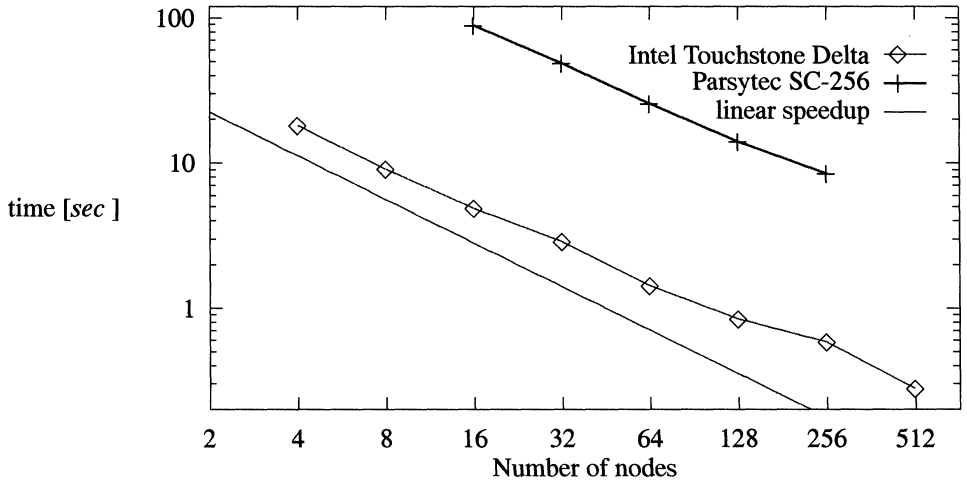

FIG. 16. *Execution time of the partition method.*

TABLE 3
Typical execution times for the production of fluid-dynamical results.

Computer	Execution time in hours on 8 nodes	64 nodes	256 nodes	512 nodes
Intel Paragon XP/S	70.8	14.3	-	-
Intel Touchstone Delta	97.2	18.2	9.1	8.4
Parsytec SC-256	-	235.2	84.2	-
Fujitsu S-600	13.1			

important factor in these periferal-but-necessary computations is the evaluation of the boundary conditions. For our application, the Fujitsu S-600 is equivalent to about 64 Paragon nodes. Whereas an individual line-relaxation step is almost four times faster on 256 Delta nodes than on 64 Delta nodes, only a factor of two is obtained in the global computation. This is due to the increasing influence of the evaluation of the boundary conditions, which introduces significant load imbalance in finer-grain computations.

6. Summary. We presented several concurrent methods for solving sets of block-tridiagonal systems on a rectangular grid. Previous studies concentrated on computations of very high granularity, which we found to be unrealistically high for our application. Therefore, we tested several methods to solve a larger number of smaller systems. All methods show a significant speedup on test runs up to 512 processes. On the Touchstone Delta, the best efficiency achieved was about 40%.

Although problem sizes will significantly increase beyond the size of the problem studied in this paper, we believe that granularity will remain of the same order of magnitude or might even decrease significantly. For most problems, the total amount of arithmetic increases faster than linearly with the total amount of data (which itself is proportional to the number of unknowns). To keep the total execution time reasonable, one should use a number of nodes proportional to the total amount of arithmetic. Hence, the amount of data per node (the granularity) is likely to decrease. There is also a numerical reason why fine-grain concurrency will become increasingly important. To improve the convergence rate, one could consider

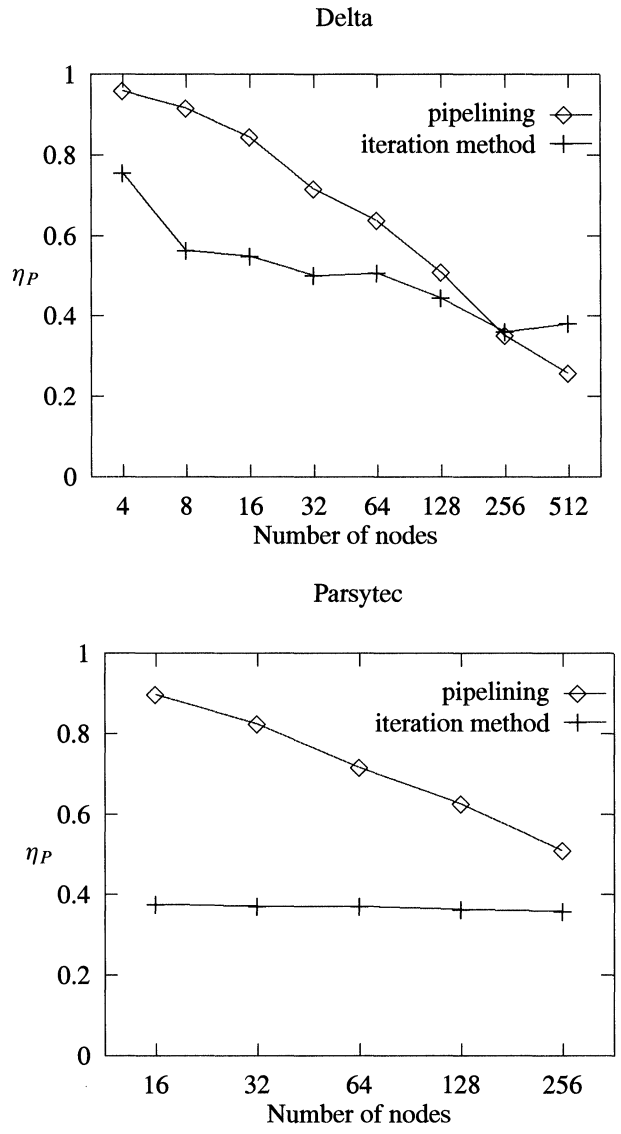

FIG. 17. *Efficiency of pipelining and iteration methods on the Delta and Parsytec multicomputers.*

using line-relaxation methods as smoothers in a nonlinear multigrid method. The coarser levels, however, quickly require computations of fine granularity.

Block-tridiagonal solvers that are efficient for fine-grain computations are difficult to implement. Nevertheless, it is possible to achieve significant execution-time improvements through the use of relatively fine-grain concurrency. A significant simplification for future endeavors is that our best results were obtained with the iterative block-tridiagonal solver, which is, from an algorithmic point of view, the simplest.

REFERENCES

[1] W. ALTHAUS, E. KRAUSE, J. HOFHAUS, AND M. WEIMER, *Bubble- and spiral-type breakdown of slender vortices*, Exp. Thermal Fluid Sci., (1995), to appear.

[2] S. BONDELI, *Divide and conquer: A parallel algorithm for the solution of a tridiagonal system of equations*, Parallel Comput., 17 (1991), pp. 419–434.
[3] M. BREUER, *Numerische Lösung der Navier–Stokes Gleichungen für dreidimensionale inkompressible instationäre Strömungen zur Simulation des Wirbelaufplatzens*, Ph.D. thesis, Aerodynamisches Institut der RWTH Aachen, Germany, 1991.
[4] M. BREUER AND D. HÄNEL, *A dual time-stepping method for 3–D, viscous, incompressible vortex flows*, Comput. & Fluids, 22 (1993), pp. 467–484.
[5] INTEL SUPERCOMPUTING SYSTEMS DIVISION, *Touchstone Delta System User's Guide*, Beaverton, Oregon, 1991.
[6] ———, *The Intel Paragon Supercomputer—an Overview*, Beaverton, Oregon, 1992.
[7] A. FROMMER, *Lösung linearer Gleichungssysteme auf Parallelrechnern*, Vieweg-Verlag, Braunschweig, Germany, 1990.
[8] C. HO AND S. JOHNSSON, *Optimizing tridiagonal solvers for alternating direction methods on Boolean cube multiprocessors*, SIAM J. Sci. Statist. Comput., 11 (1990), pp. 563–592.
[9] R. HOCKNEY, *A fast and direct solution of Poisson's equation using Fourier analysis*, J. Assoc. Comput. Mach., 1 (1965), pp. 95–113.
[10] R. HOCKNEY AND C. JESSHOPE, *Parallel Computers*, Adam Hilger, Bristol, 1981.
[11] E. ISAACSON AND H. KELLER, *Analysis of Numerical Methods*, John Wiley and Sons, New York, 1966.
[12] S. JOHNSSON, Y. SAAD, AND M. SCHULTZ, *Alternating direction methods on multiprocessors*, SIAM J. Sci. Statist. Comput., 8 (1987), pp. 686–700.
[13] W. JULING AND K. KREMER, *The massively parallel computer system of the DFG priority research program "flow simulation on supercomputers"*, in Notes on Numerical Fluid Mechanics, Vol. 38, Vieweg-Verlag, Braunschweig, Germany, 1993.
[14] A. KRECHEL, H.-J. PLUM, AND K. STÜBEN, *Parallelization and vectorization aspects of the solution of tridiagonal linear systems*, Parallel Comput., 14 (1990), pp. 31–49.
[15] H. STONE, *An efficient parallel algorithm for the solution of a tridiagonal linear system of equations*, J. Assoc. Comput. Mach., 20 (1973), pp. 27–38.
[16] H. WANG, *A parallel method for tridiagonal equations*, ACM Trans. Math. Software, 7 (1981), pp. 170–183.